

# Towards a Study of Meta-Predicate Semantics

Paulo Moura

Dep. of Computer Science, University of Beira Interior, Portugal  
Center for Research in Advanced Computing Systems, INESC–Porto, Portugal  
pmoura@di.ubi.pt

**Abstract.** We describe and compare design choices for meta-predicate semantics, as found in representative Prolog module systems and in Logtalk. We look at the consequences of these design choices from a pragmatic perspective, discussing explicit qualification semantics, computational reflection support, expressiveness of meta-predicate declarations, safety of meta-predicate definitions, portability of meta-predicate definitions, and meta-predicate performance. Our aim is to provide useful insight for debating meta-predicate semantics and portability issues based on actual implementations and common usage patterns.

**Keywords:** meta-predicate semantics, modules, objects.

## 1 Introduction

Prolog and Logtalk [1,2] meta-predicates are predicates with one or more arguments that are either goals or closures<sup>1</sup> used for constructing goals, which are called in the body of a predicate clause. Common examples are all-solutions meta-predicates such as `setof/3` and list mapping predicates. Prolog implementations may also classify predicates as meta-predicates whenever the predicate arguments need to be module-aware. Examples include the built-in database predicates such as `assertz/1` and `retract/1`.

Meta-predicates provide a mechanism for reusing programming patterns. By encapsulating meta-predicate definitions in modules and objects, exported and public meta-predicates allow client modules and client objects to reuse these patterns, customized by calls to local predicates.

In order to compare meta-predicate implementations, a number of design choices can be considered. These include explicit qualification semantics, computational reflection support, expressiveness of meta-predicate declarations, safety of meta-predicate definitions, portability of meta-predicate definitions, and meta-predicate performance.

When discussing meta-predicate semantics, it is useful to define the contexts where a meta-predicate is defined, called, and executed. The following definitions are taken from [3] and will be used in this paper:

<sup>1</sup> In Prolog and Logtalk, a closure is defined as a callable term used to construct a goal by appending one or more additional arguments.

**Definition context** This is the object or module containing the meta-predicate definition.

**Calling context** This is the object or module from which a meta-predicate is called. This can be the object or module where the meta-predicate is defined in the case of a local call or another object or module assuming that the meta-predicate is within scope.

**Execution context** This includes both the calling context and the definition context. It is comprised by all the information necessary for the language runtime to correctly execute a meta-predicate call.

In this paper, we make use of an additional definition:

**Lookup context** This is the object or module where we start looking for the meta-predicate definition. The definition can always be reexported from another module or inherited from another object.

This paper is organized as follows. Section 2 describes meta-predicate directives. Section 3 discusses the consequences of using explicit qualified meta-predicate calls and the transparency of control constructs when using explicit qualification. Section 4 describes the support for computational reflection for meta-predicates on Logtalk and Prolog module systems. Section 5 briefly presents a set of compilation safety rules for meta-predicate definitions. Section 6 discusses the portability of meta-predicate directives and meta-predicate definitions. Section 7 presents some remarks on meta-predicate performance. Section 8 summarizes our conclusions.

## 2 Meta-Predicate Directives

Meta-predicate directives are required for proper compilation of meta-predicates in both Logtalk and Prolog module systems. The design choices behind the current variations of meta-predicates directives translate to different trade-offs between simplicity and expressiveness.

### 2.1 The ISO Prolog Standard `metapredicate/1` Directive

The ISO Prolog standard for Modules [4] specifies a `metapredicate/1` directive that allows us to describe which meta-predicate arguments are normal arguments and which are meta-arguments using a predicate template. In this template, the atom `*` represents a normal argument while the atom `:` represents a meta-argument. We are not aware of any Prolog module system implementing this directive. The standard does allow for alternative meta-predicate directives, providing solely as an example a `meta/1` directive that takes a predicate indicator as argument. This alternative directive seems familiar to the `tool/2` and `module_transparent/1` directives discussed below. However, from the point-of-view of standardization and code portability, allowing for alternative directives is more harmful than helpful.

## 2.2 The Prolog `meta_predicate/1` Directive

The ISO Prolog specification of a meta-predicate directive suffers from major shortcomings [5]. First, it is not possible to distinguish between goals and closures. Second, it is not possible to represent the instantiation mode of the normal arguments (the instantiation mode of meta-arguments is implicit).

The de facto standard solution for specifying closures is to use a non-negative integer representing the required number of additional arguments [6]. By interpreting a goal as a closure requiring zero additional arguments, we can reserve the atom `:` to represent arguments that need to be module (or object) aware without necessarily referring to a predicate. This convention is found in recent SICStus Prolog [7] and SWI-Prolog versions [8] and is being adopted by other Prolog compilers. In Prolog module systems where module expansion only needs to distinguish between normal arguments and meta-arguments, using an integer for representing closures can still be useful for cross-reference tools.

For representing the instantiation mode of normal arguments, the atoms `+`, `?`, `@`, and `-` are commonly used, as specified in the ISO Prolog standard [9].

Despite the level of detail in the description of meta-predicate arguments, there is, however, a known representation shortcoming. Some predicates accept a list of options where one or more options are module-aware. For example, the third argument of the predicate `thread_create/3` [10] is a list of options that can include an `at_exit/1` option. This option specifies a goal to be executed when a thread terminates. In this case, the argument is not a meta-argument but may *contain* a sub-term that will be used as a meta-argument. Although we could devise (a most likely cumbersome) syntax for these cases, the elegant solution for this representation problem is provided by the `tool/2` and `module_transparent/1` directives discussed below.

## 2.3 The Logtalk `meta_predicate/1` Directive

Logtalk uses a `meta_predicate/1` directive with the atom `:` replaced by `::` for consistency with the message sending operator and allowing `::` to be synonym to the integer zero to indicate goal arguments. As in the Prolog directive described above, closures are represented by a non-negative integer. Logtalk uses this information to verify meta-predicate definitions, as discussed in [3]. Logtalk supports a `mode/2` predicate directive for specifying the instantiation mode and the type of predicate arguments (plus the predicate determinism). Therefore, the atom `*` is used to indicate normal arguments in `meta_predicate/1` directives.

## 2.4 The `tool/2` and `module_transparent/1` Directives

An alternative, used in ECLiPSe [11] and in earlier SWI-Prolog versions [12] is to simply declare meta-predicates as *module transparent*, forgoing the specification of which arguments are normal arguments and which arguments are meta-arguments. For this purpose, ECLiPSe provides a `tool/2` directive and SWI-Prolog provides a (now deprecated) `module_transparent/1` directive. These

directives take predicate indicators as arguments and thus support a simpler and user-friendlier solution when compared with the `meta_predicate/1` directive. However, we have show in [3] that distinguishing between goals and closures and specifying the exact number of closure additional arguments is necessary to avoid misuse of meta-predicate definitions.

### 3 Explicit Qualification Semantics

The semantics of explicit qualification is the single most significant design decision on meta-predicate semantics. This section compares two different semantics, found on actual implementations, for the explicit qualification of meta-predicate and control constructs.

#### 3.1 Explicit Qualification of Meta-Predicate Calls

Given an explicit qualified meta-predicate call, we have two sensible choices for the corresponding semantics:

1. The explicit qualification sets only the initial lookup context for the meta-predicate definition. Therefore, all meta-arguments that are not explicitly-qualified are called in the meta-predicate calling context.
2. The explicit qualification sets both the initial lookup context for the meta-predicate definition and the meta-predicate calling context. Therefore, all meta-arguments that are not explicitly-qualified are called in the meta-predicate lookup context (usually the same as the meta-predicate definition context).

These two choices for explicit qualification semantics are also described in the ISO Prolog standard for modules. This standard specifies a read-only flag, `colon_sets_calling_context`, which would allow a programmer to query the semantics of a particular module implementation.

Logtalk and the ECLiPSe module system implement the first choice. Prolog module systems derived from the Quintus Prolog module system [6], including those found on Ciao Prolog, SICStus Prolog, SWI-Prolog, and YAP implement the second choice (the native XSB module system is atom-based, not predicate-based; we will not discuss it here).

In order to illustrate the differences between the two choices above, consider the following example, running on Prolog module systems implementing the second choice. First, we define a meta-predicate library:

```
:- module(library, [my_call/1]).

:- meta_predicate(my_call(0)).
my_call(Goal) :-
    write('Calling: '), writeq(Goal), nl, call(Goal).

me(library).
```

Second, we define a simple client module:

```
:- module(client, [test/1]).

:- use_module(library, [my_call/1]).

test(Me) :-
    my_call(me(Me)).

me(client).
```

To test our code, we use the following query:

```
?- client:test(Me).

Calling: client:me(_)
Me = client
yes
```

This query provides the expected result. But consider the following seemingly innocuous changes to the client module:

```
:- module(client, [test/1]).

test(Me) :-
    library:my_call(me(Me)).

me(client).
```

In this second version, we use explicit qualification in order to call the `my_goal/1` meta-predicate. Repeating our test query gives:

```
?- client:test(Me).

Calling: library:me(_)
Me = library
yes
```

In order for a programmer to understand this result, we need to be aware that the `:/2` operator both calls a predicate in another module and changes the calling context of the predicate to that module. The first use is expected. The second use is not obvious, is counterintuitive (due to different semantics between implicitly-qualified and explicitly-qualified calls to the same predicate), and often not properly documented. In the common case where we are reusing a library meta-predicate, the user (rightfully) expects that a local predicate will be called when using a meta-argument with the same functor and arity instead of a predicate with the same name in the meta-predicate definition context. Indeed, it is unlikely that a predicate with the same name of the local predicate even exist in the library module. We can, however, conclude that the meta-predicate definition is still working as expected as the calling context is set to the library module. If we still want the `me/1` predicate to be called in the context of the client module instead, we need to explicitly qualify the meta-argument by writing:

```
test(Me) :-
    library:my_call(client:me(Me)).
```

This is an awkward solution but it works as expected in the rare cases that require explicit qualification. It should be noted that the idea of the `meta_predicate/1` directive is to avoid the need for explicit qualifications in the first place. But that requires using `use_module/1-2` directives for importing the meta-predicates and implicit qualification when calling them. This explicit qualification of meta-arguments is not necessary in Logtalk and in the ECLiPSe module system, where explicit qualification of a meta-predicate call sets where to start looking for the meta-predicate definition, not where to look for the meta-arguments definition.

The semantics of the `:/2` operator in Prolog module systems derived from the Quintus Prolog module system is probably rooted in optimization goals. When a directive `use_module/1` is used, most (if not all) Prolog compilers require the definition of the imported module to be available (thus resolving the call at compilation time). However, that does not seem to be required when compiling an explicitly qualified module call. For example, using recent versions of SICStus Prolog, SWI-Prolog, and YAP, the following code compiles without errors or warnings (despite the fact that the module `fictitious` does not exist):

```
:- module(client, [test/1]).

test(X) :-
    fictitious:predicate(X).
```

Thus, in this case the `fictitious:predicate/1` call is resolved at runtime. In our example above with the explicit call to the `my_call/1` meta-predicate, the implementation of the `:/2` operator propagates at runtime the module prefix to the meta-arguments that are not explicitly qualified. This runtime propagation translates to a performance penalty. Therefore, and not surprisingly, the use of explicit qualification is discouraged by the Prolog implementers. In fact, until recently, most Prolog implementations provided poor performance for `:/2` calls even when the necessary module information was available at compile time.

### 3.2 Transparency of Control Constructs

One of the design choices regarding meta-predicate semantics is the transparency of control constructs to explicit qualification. The relevance of this topic is that most control constructs can also be regarded as meta-predicates. In fact, there is a lack of agreement on the Prolog community on which language elements are control constructs and which language elements are predicates. For the purposes of our discussion, we use the classification found on the ISO Prolog standard, which specifies the following control constructs: `call/1`, *conjunction*, *disjunction*, *if-then*, *if-then-else*, and `catch/3`. The standard also specifies `true/0`, `fail/0`, `!/0`, and `throw/1` as control constructs but none of these can be interpreted as a meta-predicate.

When a control construct is transparent to explicit qualification, the qualification propagates to all the control constructs arguments that are not explicitly qualified. For example, the following equivalences hold:

$$\begin{aligned} M:(A, B) &\Leftrightarrow (M:A, M:B) \\ M:(A; B) &\Leftrightarrow (M:A; M:B) \\ M:(A \rightarrow B; C) &\Leftrightarrow (M:A \rightarrow M:B; M:C) \end{aligned}$$

In Prolog module systems where the `:/1` operator sets both the meta-predicate lookup context and the meta-arguments calling context, the above equivalences are consistent with the explicit qualification semantics of meta-predicates described in the previous section. For example:

$$\begin{aligned} M:\text{findall}(T, G, L) &\Leftrightarrow \text{findall}(T, M:G, L) \\ M:\text{assertz}(A) &\Leftrightarrow \text{assertz}(M:A) \end{aligned}$$

This is also true for user-defined meta-predicates. For the example presented in the previous section, the following equivalence holds:

$$\text{library}:\text{my\_call}(\text{me}(\text{Me})) \Leftrightarrow \text{my\_call}(\text{library}:\text{me}(\text{Me}))$$

We can conclude that the different semantics of implicitly and explicitly qualified meta-predicate calls, which is at odds with most user expectations, allows the semantics of explicitly qualified control constructs to be consistent with the semantics of explicitly qualified meta-predicate calls.

In systems such as ECLiPSe or Logtalk, where explicit qualification only sets the lookup context, the semantics of control constructs and meta-predicates are different. In Logtalk, the following equivalences for control constructs are handy, supported, and can be interpreted as a shorthand notation for sending a set of messages to the same object (the `:/2` operator is used in Logtalk for message sending):

$$\begin{aligned} O::(A, B) &\Leftrightarrow (O::A, O::B) \\ O::(A; B) &\Leftrightarrow (O::A; O::B) \\ O::(A \rightarrow B; C) &\Leftrightarrow (O::A \rightarrow O::B; O::C) \end{aligned}$$

ECLiPSe implements a simpler design choice, disallowing the above shorthands, and thus treating control constructs and meta-predicates uniformly. Both Logtalk and ECLiPSe provide, however, the same syntax for implicitly and explicitly qualified meta-predicate calls. Consider the following objects, corresponding to a a Logtalk version of the Prolog module example used in the previous section:

```
:- object(library).

:- public(my_call/1).
:- meta_predicate(my_call(:)).
my_call(Goal) :-
    write('Calling: '), writeq(Goal), nl,
```

```

        call(Goal),
        sender(Sender), write('Sender: '), writeq(Sender).

    me(library).

:- end_object.

:- object(client).

    :- public(test/1).
    test(Me) :-
        library::my_call(me(Me)).

    me(client).

:- end_object.

```

Our test query becomes:

```

?- client::test(Me).

Calling: me(_G216)
Sender: client
Me = client.
yes

```

That is, meta-arguments are always called in the context of the meta-predicate call. Logtalk also implements common built-in meta-predicates such as `call/1-N`, `\+/1`, `findall/3`, and `phrase/3` with the same semantics as user-defined meta-predicates. In order to avoid misinterpretations, these built-in meta-predicates are implemented as private predicates.<sup>2</sup> Thus, the following call is illegal and results in a permission error:

```

?- some_object::findall(T, g(T), L).

error(
    permission_error(access,private_predicate,findall(T,g(T),L)),
    some_object::findall(T,g(T),L),
    user)

```

The correct call would be:

```

?- findall(T, some_object::g(T), L).

```

---

<sup>2</sup> Logtalk supports *private*, *protected*, and *public* predicates. A predicates may also be *local* if no scope directive is present, making the predicate invisible to the built-in reflection predicates.



We can conclude that ensuring the same semantics for implicitly and explicitly qualified meta-predicate calls requires different semantics for explicitly qualified control constructs, and thus a clear distinction between control constructs and predicates, in order to provide a reading for explicitly qualified control constructs that matches user expectations. This distinction can be rendered moot, however, if we simply disallow explicit qualification of control constructs, as exemplified by ECLiPSe.

## 4 Computational Reflection Support

Computational reflection allows us to perform computations about the *structure* and the *behavior* of an application. In the case of meta-predicates, structural reflection allows us to find where the meta-predicate is defined and about the meta-predicate template, while behavioral reflection allows us to access the meta-predicate execution context. As described in Section 1, a meta-predicate execution context includes information about from where the meta-predicate is called. This is only meaningful, however, in the presence of a predicate encapsulation mechanism such as modules or objects. Access to the execution-context is usually not required for common user-level meta-predicate definitions but can be necessary when meta-predicates are used, for example, to extend Prolog or Logtalk meta-call features. In the case of Logtalk, full access to predicate execution context is provided by the `sender/1`, `self/1`, `this/1`, and `parameter/2` built-in predicates [2]. For Prolog compilers supporting modules, the following table provides an overview of the reflection built-in predicates that can be used to access a meta-predicate execution context:

Prolog compiler	Built-in reflection predicates
Ciao 1.10	<code>predicate_property/2</code> (in library <code>prolog_sys</code> )
ECLiPSe 6.0	<code>get_flag/3</code>
SICStus Prolog 4.1	<code>predicate_property/2</code>
SWI-Prolog 5.9.10	<code>context_module/1</code> , <code>predicate_property/2</code> , <code>strip_module/3</code>
XSB 3.2	<code>predicate_property/2</code>
YAP 6.0	<code>context_module/1</code> , <code>predicate_property/2</code>

From this table we conclude that the only built-in predicate common to these Prolog compilers is `predicate_property/2`. Together with the ECLiPSe `get_flag/3` and the SWI-Prolog and YAP `context_module/1` predicates, these built-ins only provide *structural* reflection. Specifically, information about the meta-predicate template and the definition context of the meta-predicate. SWI-Prolog is the only compiler that provides *built-in* access to the meta-predicate calling context using the predicate `strip_module/3`. As a simple example of using this predicate consider the following module:

```
:- module(m, [mp/2]).

:- meta_predicate(mp(0, -)).
```

```

mp(Goal, Caller) :-
    strip_module(Goal, Caller, _),
    call(Goal).

```

After compiling and loading this module, the following queries illustrate both the functionality of the `strip_module/3` predicate and the consequences of explicit qualification of the meta-predicate call:

```

?- mp(true, Caller).
Caller = user.

?- m:mp(true, Caller).
Caller = m.

```

For similar Prolog compiler module systems, descending from the Quintus Prolog module system, it is possible to access the meta-predicate calling context by looking into the implicit qualification of a meta-argument:

```

:- module(m, [mp/2]).

:- meta_predicate(mp(0, -)).

mp(Goal, Caller) :-
    Goal = Caller:_,
    call(Goal).

```

After compiling and loading this module, we can reproduce the results illustrated by the queries above for the SWI-Prolog version of this module. The only possible caveat would be if the Prolog compiler fails to ensure that there is always a single qualifier for a goal. That is, that terms such as `M1:(M2:(M3:G))` are never generated internally when propagating module qualifications.

In the case of ECLiPSe, a built-in predicate for accessing the meta-predicate calling context is not necessary as the `tool/2` directive works by connecting a meta-predicate interface with its implementation:

```

:- module(m).

:- export(mp/2).
:- tool(mp/2, mp/3).

mp(Goal, Caller, Caller) :-
    call(Goal).

```

After compiling and loading this module, repeating the above queries illustrate the difference in explicit qualification semantics between ECLiPSe and the other Prolog compilers:

```

[eclipse 16]: mp(true, Caller).

Caller = eclipse
Yes (0.00s cpu)
[eclipse 17]: m:mp(true, Caller).

Caller = eclipse
Yes (0.00s cpu)

```

Note that the module `eclipse` is the equivalent of the module `user` in other compilers.

## 5 Secure Meta-Predicate Definitions

Meta-predicate definitions should not provide a mechanism for calling client predicates other than the ones intended by the meta-predicate calls. This, however, is mostly meaningful for languages such as Logtalk and for Prolog module systems, such as ECLiPSe and Ciao [13], that aim to enforce object and module predicate scope rules. The following set of compilation rules, described and illustrated in detail in [3], contribute to make meta-predicate definitions secure:

1. The meta-arguments of a meta-predicate clause head must be variables.
2. Meta-calls whose arguments are not variables appearing in meta-argument positions in the clause head must be compiled as calls to local predicates.
3. Meta-predicate closures must be used within a `call/2-N` built-in predicate call that complies with the corresponding meta-predicate directive.

These rules are implemented in Logtalk. For Prolog module systems whose design allows any module predicate to be called using explicit module qualification, these rules may be regarded as best practice for writing meta-predicates and thus useful for checking meta-predicate definitions for possible errors (e.g. as part of lint checkers).

## 6 Meta-predicate Definitions Portability

The portability of meta-predicate definitions depends on two main factors: portability of the meta-predicate directives and portability of the meta-call primitives used when implementing the meta-predicates. Other factors that may impact portability are the preprocessing solutions for improving meta-predicate performance, described in Section 7, and the mechanisms for computational reflection about meta-predicate definition and execution, discussed in Section 4.

## 6.1 Specification of Closures and Instantiation Modes in Meta-Predicate Directives

The main portability issue of meta-predicate directives is the use of non-negative integers to specify closures and the atoms used to specify the instantiation mode of normal arguments.

Although the use of non-negative integers comes from Quintus Prolog, it was mostly regarded as a way to provide information to cross-reference and documentation tools. Prolog compilers such as SICStus Prolog [7] and YAP [14] accept this notation but only for compatibility with existing code. Other Prolog compilers such as Ciao define alternative but incompatible syntaxes for specifying closures.

There is also some variation in the atoms used for representing the instantiation modes of normal arguments. Some Prolog compilers use an extended set of atoms for documenting argument instantiation modes compared to the basic set (+, ?, @, and -) found, for example, in the ISO Prolog standard. It is therefore tempting to use these extended sets in meta-predicate directives, which will likely raise portability issues.

We hope that recent Prolog standardization initiatives, specially the development of portable libraries, will lead to establish a de facto standard meta-predicate directive derived from the extended directive described in Section 2.

## 6.2 The `call/1-N` Control Constructs

The `call/1` control construct is specified in the ISO Prolog standard [9]. This control construct is implemented by virtually all Prolog compilers. The `call/2-N` control constructs, whose use is strongly recommended whenever a meta-predicate works with closures (see Section 5), is specified in the ISO Prolog standard Core Revision proposal [15]. A growing number of Prolog compilers implement these control constructs but with different maximum values for `N`, which can raise some portability problems. Ideally, the implementation of the `call/1-N` control constructs would support `N` up to the maximum predicate arity. However, the feasibility of supporting a large value for `N` depends on the design decisions of a Prolog compiler implementation. It should also be noted that, in some Prolog compilers such as recent SWI-Prolog and YAP versions, the maximum predicate arity of a Prolog compiler is unbounded. From a language design point-of-view, limiting the maximum value of `N` to a value different from maximum term arity can be interpreted as a flaw. It certainly seems odd that a programmer can use the `=../2` built-in predicate and the `call/1` control construct to build and call a goal but that a similar solution cannot be used by the Prolog implementer when compiling `call/2-N` calls. From a pragmatic point-of-view, it is unlikely that user written code (not necessarily user *generated* code) would require a large upper limit of `N`. Despite the apparent lack of agreement, the more significant portability issues here are Prolog compilers only supporting `call/1` or a arguably small value of `N`. The following table summarizes the implementations of the `call/2-N` control construct on selected Prolog compilers:

System	N	Notes
B-Prolog 7.4	10/65535	(interpreter/compiler i.e. maximum arity)
Ciao 1.10	255	(maximum arity using the <code>hiord</code> library)
CxProlog 0.94.0	9	—
ECLiPSe 6.0	1	—
GNU Prolog 1.3.1	11	—
JIProlog 3.0.2	5	—
K-Prolog 6.0.4	9	—
Qu-Prolog 8.10	1	(supports a <code>call_predicate/1-5</code> built-in predicate)
SICStus Prolog 4.1	255	(maximum arity)
SWI-Prolog 5.9.10	8	( <code>meta_predicate/1</code> directive limit)
XSB 3.2	11	—
YAP 6.0	12	—

This table only lists *built-in* support for `call/2-N` control construct. While this control construct can be defined by the programmer using the built-in predicate `=../2` and an `append/3` predicate, such definitions provide relative poor performance due to the construction and appending of temporary lists of arguments.

## 7 Meta-Predicate Performance

Considering that meta-programming is often touted as a major feature of Prolog, the relative poor performance of meta-calls is embarrassing. For those cases where performance is an important factor, the usual solution is to interpret meta-predicate definitions as high-level macros and to preprocess meta-predicate calls in order to replace them with calls to auxiliary predicate definitions that do not contain meta-calls. This preprocessing is usually only performed on code marked as stable as the auxiliary predicates often complicate debugging. The preprocessing code is often implemented in optional libraries, which can be found on several Prolog compilers such as ECLiPSe, SWI-Prolog, and YAP. These libraries, however, require custom code for each meta-predicate. Therefore, user-defined meta-predicates will fail to match the performance of library-supported meta-predicates unless the user also writes its own custom preprocessing code. A more generic solution for preprocessing meta-predicate definitions is needed to make using these programming patterns more appealing for performance-critical applications.

## 8 Conclusions

We presented a comprehensive set of meta-predicate design decisions based on current practice in Logtalk and Prolog module systems. The most remarkable result is that none of the two commonly implemented semantics for explicitly qualified calls provides an ideal solution that both matches user expectations and

allows the distinction between predicates and control constructs to be waived. By describing the consequences of these design decisions we provided useful insight to discuss meta-predicate semantics, often a difficult subject for inexperienced programmers. We hope that this paper contributes to a convergence of meta-predicate directive syntax, meta-predicate semantics, and meta-predicate related reflection built-in predicates among Prolog module systems.

**Acknowledgements.** We are grateful to Ulrich Neumerkel, Jan Wielemaker, and Richard O’Keefe for interesting discussions about explicitly-qualified meta-predicate call semantics on the SWI-Prolog mailing list. We thank also the anonymous reviewers for their informative comments. This work is partially supported by FCT project MOGGY – PTDC/EIA/70830/2006.

## References

1. Moura, P.: Logtalk - Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
2. Moura, P.: Logtalk 2.39.0 User and Reference Manuals. (February 2010)
3. Moura, P.: Secure Implementation of Meta-predicates. In Gill, A., Swift, T., eds.: Proceedings of the Eleventh International Symposium on Practical Aspects of Declarative Languages. Volume 5418 of Lecture Notes in Computer Science., Berlin Heidelberg, Springer-Verlag (January 2009) 269–283
4. ISO/IEC: International Standard ISO/IEC 13211-2 Information Technology — Programming Languages — Prolog — Part II: Modules. ISO/IEC (2000)
5. O’Keefe, R.: An Elementary Prolog Library. <http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm>
6. Swedish Institute for Computer Science: Quintus Prolog User’s Manual (Release 3.5). Swedish Institute for Computer Science. (December 2003)
7. Swedish Institute for Computer Science: SICStus Prolog 4.1 User Manual. (2009)
8. Wielemaker, J.: SWI-Prolog 5.9 Reference Manual. (March 2010)
9. ISO/IEC: International Standard ISO/IEC 13211-1 Information Technology — Programming Languages — Prolog — Part I: General core. ISO/IEC (1995)
10. (editor), P.M.: ISO/IEC DTR 13211–5:2007 Prolog Multi-threading predicates. <http://logtalk.org/plstd/threads.pdf>
11. Cheadle, A.M., Harvey, W., Sadler, A.J., Schimpf, J., Shen, K., Wallace, M.G.: ECLiPSe: A tutorial introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College, London (2003)
12. Wielemaker, J.: An overview of the SWI-Prolog programming environment. In Mesnard, F., Serebenik, A., eds.: Proceedings of the 13th International Workshop on Logic Programming Environments, Heverlee, Belgium, Katholieke Universiteit Leuven (December 2003) 1–16 CW 371.
13. Gras, D.C., Hermenegildo, M.V.: A New Module System for Prolog. In: CL’00: Proceedings of the First International Conference on Computational Logic, London, UK, Springer-Verlag (2000) 131–148
14. Costa, V.S.: The YAP User’s Manual: version 6.0. (2010)
15. Moura, P.: ISO/IEC DTR 13211–1:2006 New built-in flags, predicates, and functions proposal. <http://logtalk.org/plstd/core.pdf> (October 2009)